Static Verification of Non-Functional Software Requirements in the ISO-26262

Daniel Kästner, Christian Ferdinand

AbsInt GmbH
Science Park 1
66123 Saarbrücken
kaestner@absint.com
ferdinand@absint.com

Abstract: The norm ISO-26262 aims at ascertaining the functional safety of Automotive Electric/Electronic Systems. It is not focused on purely functional system properties, but also demands to exclude nonfunctional safety hazards in case they are critical for a correct functioning of the system. Examples are violations of timing constraints in real-time software and software crashes due to runtime errors or stack overflows. The ISO-26262 ranks the static verification of program properties among the prominent goals of the software design and implementation phase. Static program analyzers are available that can prove the absence of certain non-functional programming errors, including those mentioned above. Static analyzers can be applied at different stages of the development process and can be used to complement or replace dynamic test methods. This article gives an overview of static program analysis techniques focusing on non-functional program properties, investigates the non-functional requirements of the ISO-26262 and discusses the role of static analyzers in the ISO-26262.

1 Introduction

The norm ISO-26262 [ISO11a] is based on the Functional Safety Standard IEC 61508 [IEC10] and aims at ascertaining the functional safety of Automotive Electric/Electronic Systems. It has been published as an international standard in August 2011, replacing the IEC 61508 as formal legal norm for road vehicles.

The ISO-26262 is not focused on purely functional system properties, but also demands to exclude nonfunctional safety hazards in case they are critical for the correct functioning of the system. Examples are violations of timing constraints in real-time software and software crashes due to runtime errors or stack overflows. Depending on the criticality level of the software the absence of safety hazards has to be demonstrated by formal methods or testing with sufficient coverage. This also holds for related safety standards like DO-178B [Rad], DO-178C, IEC-61508 [IEC10], and EN-50128 [CEN09]. The ISO-26262 demands to consider the non-functional safety goals throughout the entire development process: from specification, architectural design and implementation to validation and testing.

While the ISO-26262 does not enforce specific testing and verification methods, the im-

portance of static verification is emphasized: it is considered one of the three goals of the software unit design and implementation stage. The advantage of static techniques is that they do not depend on specific test cases but provide results which are only based on the source (or binary) code. Testing, in general, cannot show the absence of errors [Rad] since only a concrete set of inputs is considered and usually no full test coverage can be achieved.

The term *static analysis* is used to describe a variety of program analysis techniques with the common property that the results are only based on the software structure. The most advanced static analysis technique is the so-called *Abstract Interpretation* [CC77] which is counted to the formal methods. Abstract Interpretation is a semantics-based methodology for program analysis. It provides results that are valid for all program runs with all inputs, thus enabling full control and data coverage. Abstract Interpretation-based tools are in industrial use that can prove the absence of stack overflows, compute safe upper bounds on the worst-case execution time [SLH+05], and prove the absence of runtime errors [DS07].

Specifically for non-functional program properties like timing, stack size, and runtime errors dynamic testing is difficult since identifying safe end-of-test criteria is an unsolved problem. In consequence the required test effort is high, the tests require access to the physical hardware and the results are not complete. Let's consider timing as an example: With modern processors the timing behavior of an instruction depends on the instructions previously executed, e.g., due to cache or pipeline effects. In consequence even MC/DC coverage is not enough to determine worst-case execution time information. This is because the code can be thoroughly tested in multiple runs without ever having exercised the longest path on any single run. Since different execution paths cannot be distinguished there is no control which paths have been covered. In contrast, Abstract Interpretation-based static analyses can be seen as equivalent to testing with full coverage. Thus, in the areas where validation by static analysis is technically feasible and applied in industry it defines the state-of-the-art testing technology. As an example the static WCET analyzer aiT [SLH+05] has been used by NASA in the Toyota Motor Corporation Unintended Acceleration Investigation to investigate the timing behavior, and could show the absence of timing-related software defects [NAS11].

In the following we will give an overview of the various different static analysis techniques focusing on Abstract Interpretation (Sec. 2). Sec. 3 discusses the predominant non-functional safety goals: timing correctness, absence of stack overflows, and absence of runtime errors. The application of static analysis techniques to demonstrate these goals is shortly sketched. The main part of the article is Sec. 4 which tracks the non-functional requirements through the different sections of Chap. 6 of ISO-26262 [ISO11b]. Sec. 5 discusses the integration of static analysis tools in the development process, focusing on interfaces, tool couplings, and tool qualification. Sec. 6 concludes.

2 Static Analysis Techniques

Static program analyzers compute information about the software under analysis without actually executing it. The analyzers can work at the source code level, or at the object or executable code level. The term "static analyzer" is very broadly used and can denote a variety of different techniques. This section gives an overview based on [CCF+07].

Sometimes the term "static analyzer" is understood as including style checkers looking for deviations from coding style rules (like Safer C checking for C fault and failure modes [Hat95] or MISRA C checker [Mot04]). Such syntactical methods are not "semantics-based", and thus, e.g., cannot determine variable values or check for correct runtime behavior.

Semantics-based static analyzers use an explicit (or implicit) program semantics that is a formal (or informal) model of the program executions in all possible or a set of possible execution environments. Based on the program semantics, information about data and control flow is obtained. The most important characteristics of static analyzers is whether they are *sound* or *unsound*. A static analyzer is called *sound* if the computed results hold for any possible program execution.

A program analyzer is unsound when it can omit to signal an error that can appear at runtime in some execution environment. Unsound analyzers are *bug hunters* or *bug finders* aiming at finding some of the bugs in a well-defined class. Their main defect is unreliability, being subject to false negatives thus claiming that they can no longer find any bug while many may be left in the considered class.

Unsoundness may be caused by

- discarding program effects:
  A simple example is a write access through a pointer, e.g. `*p=42`. If the pointer analysis is imprecise and concludes that `p` may point to 1000 variables then the only sound way to proceed is to assume that all 1000 variables are modified by this assignment and may take the value 42. Of course, this would make the result very imprecise. Hence, some unsound analyzers may decide to ignore instructions on which they know they probably are very imprecise. As an example the source code analyzer CMC [ECH+01] from Coverity ignores code with complex semantics whose analysis would be too hard to design. Some C compilers can produce stack usage estimates but they fail to take the effect of inline assembly code or certain library routines into account, thus possibly underestimating the maximal stack usage.

- ignoring some types of errors:
  The source code analyzer UNO concentrates exclusively on uninitialized variables, nil-pointer references, and out-of-bounds array indexing [Hol02]. Another example is bug pattern mining (cf. e.g., Klocwork K7™ [Klo]), looking for just a few key types of programmer errors. Due to high false alarm rates, also alarm filtering often is applied, e.g., Splint [LE01] sorts the most probable messages according to common programming practices.

- disregarding some runtime executions:
  Some static stack analyzers ignore recursions and only count the stack usage of one single execution of a recursive function. Dynamic analyzers, i.e., test & measurement approaches are unsound since it is impossible to explore all possible executions of a program on a machine.

- changing the program semantics:
  Source code analyzers have to take the machine representation of numbers into account. It is not correct to assume that integers are in $\mathbb{Z}$, and reals in $\mathbb{R}$. Instead the limited bitwidth of machine numbers in general, and especially floating-point rounding errors have to be taken into account.

2.1 Abstract Interpretation

The set of all possible runtime executions of a program is defined by the so-called concrete semantics. Unfortunately most interesting program properties are undecidable in the concrete semantics. The theory of abstract interpretation [CC77] offers a semantics-based methodology for static program analyses where the concrete semantics is mapped to a simpler abstract model, the so-called abstract semantics. The static analysis is computed with respect to that abstract semantics. Compared to an analysis of the concrete semantics, the analysis result may be less precise but the computation may be significantly faster. By skilful definition of the abstract semantics a suitable trade-off between precision and efficiency can be obtained. Abstract interpretation supports formal correctness proofs: it can be proved that an analysis will terminate and that it is sound, i.e., that it computes an overapproximation of the concrete semantics. Moreover it can be shown that imprecisions always occur on the safe side. Examples of such proofs can be found in [Fer97, The04, Min04].

Let's illustrate this with two application scenarios: In runtime error analysis, soundness means that the analyzer never omits to signal an error that can appear in some execution environment. If no potential error is signalled, definitely no runtime error can occur, i.e., there are *no false negatives*. If a potential error is reported, the analyzer cannot exclude that there is a concrete program execution triggering the error. If there is no such execution, this is a false alarm (*false positive*). This imprecision is on the safe side: it can never happen that there is a runtime error which is not reported. In WCET analysis, soundness means that the computed WCET bound holds for any possible program execution. Safety means that the only imprecision occurring is overestimation: the WCET must never be underestimated. Abstract Interpretation allows these soundness and safety properties to be formally proven.

3 Non-Functional Safety Goals

Safety standards like ISO-26262 [ISO11a], DO-178B [Rad], DO-178C, IEC-61508 [IEC10], and EN-50128 [CEN09] require to identify functional and non-functional hazards and to demonstrate that the software does not violate the relevant safety goals. Some non-functional safety hazards can be critical for the correct functioning of the system: violations of timing constraints in real-time software and software crashes due to runtime errors or stack overflows. Proving their absence is explicitly listed as a verification goal by all of the above mentioned standards. In this chapter we will describe the background and the challenges of demonstrating the absence of these non-functional safety hazards.

3.1 Absence of Stack Overflow

A possible cause of catastrophic failure is a stack overflow which might cause the program to behave in a wrong way or to crash altogether. When they occur, stack overflows can be hard to diagnose and hard to reproduce. The problem is that the memory area for the stack usually must be reserved by the programmer. Underestimation of the maximum stack usage leads to stack overflow, while overestimation means wasting memory resources. Measuring the maximum stack usage with a debugger is no solution since one only obtains a result for the selected set of program runs with fixed inputs. Even many repeated measurements with various inputs cannot guarantee that the maximum stack usage is ever observed.

3.2 Worst-Case Execution/Response Time Guarantees

Many tasks in safety-critical embedded systems have hard real-time characteristics. Failure to meet deadlines may be as harmful as producing wrong output or failure to work at all. Timing verification implies a reliable verification of code execution times, response times and end-to-end latencies to find the critical corner-case scenarios of system execution in the presence of complex processor architectures, multi-tasking operating systems, and network protocols. First, safe upper bounds for the execution times (WCETs) of non-interrupted tasks have to be determined. Then the worst-case response times (WCRTs) of an entire system from the task WCETs and information about possible interrupts and their priorities have to be computed [KFH+11]. In the following we will focus on determining the WCET of a task; for more information about system-level WCRT analysis see [HHJ+05].

Modern hardware makes computing safe and precise WCET bounds a challenge. There is typically a large gap between the cycle times of modern microprocessors and the access times of main memory. Caches and branch target buffers are used to overcome this gap in virtually all performance-oriented processors (including high-performance microcontrollers and DSPs). Pipelines enable acceleration by overlapping the executions of

different instructions. Consequently the execution behavior of the instructions cannot be analyzed separately since it depends on the execution history. Cache memories usually work very well, but under some circumstances minimal changes in the program code or program input may lead to dramatic changes in cache behavior. For (hard) real-time systems, this is undesirable and possibly even hazardous. Making the safe yet – for the most part – unrealistic assumption that all memory references lead to cache misses results in the execution time being overestimated by several hundred percent.

The widely used classical methods of predicting execution times are not generally applicable. Software monitoring and dual-loop benchmark modify the code, which in turn changes the cache behavior. Hardware simulation, emulation, or direct measurement with logic analyzers can only determine the execution time for some fixed inputs. They cannot be used to infer the execution times for all possible inputs in general.

Static analysis by Abstract Interpretation is a safe method for timing analysis providing guaranteed upper bounds for the worst-case execution time (WCET) of tasks [SLH+05]. Static analyzers are available for complex processors and yield precise results. However, a basic requirement is that the timing behavior of the processor is *predictable*. In general, predictability degrades with increasing interference by accesses to shared resources. In single-core processors this can be observed with speculative hardware mechanisms like caches, out-of-order pipelining, or branch prediction. If a task $T1$ on a processor with caches is preempted by another task, the memory blocks needed by $T1$ might be evicted from the cache. If $T1$ continues after the preemption additional cache misses can be the consequence. Covering all potential preemption scenarios by measurements clearly is infeasible. Cache-related preemption costs can be considered by static analysis, but the difference between average and worst-case execution time grows. By cache partitioning or cache locking predictability can be improved and the potential variance of execution time decreases.

On multi-core processors not only the interferences within each core have to be considered. Now there are additional interferences due to concurrent accesses to shared resources by different cores. Such interferences may be caused by accesses to common caches, common memory banks, or common FLASH or prefetch buffers. For a given multi-core architecture potential interferences have to be carefully examined. Then a configuration can be determined which enables or facilitates predictable performance [CFG+10].

3.3 Absence of Runtime Errors

Another important goal when developing critical software is to prove that no runtime errors can occur. Examples for runtime errors are division by zero, invalid pointer accesses, array bound violations, or arithmetic overflows. A well-known example for the possible effects of runtime errors is the explosion of the Ariane 5 rocket in 1996 [Lea96]. As detailed above, software testing can be used to detect errors, but not to prove their absence. E.g. to reliably detect a division by zero, all potential values of each variable of the denominator would have to be exerted.

The result of a sound static runtime error analysis for a statement x will be either "(i) statement x will not cause an error", or "(ii) statement x may cause an error". In the first case, the user can rely on the absence of errors, in the second case either an error has been found, or there was a false alarm (false positive). The imprecision induced by the semantical abstraction allows to compute results in acceptable time, even for large software projects. Nevertheless the results are reliable, i.e., the analysis will only err on the safe side: if the analyzer does not detect any error, the absence of errors has been proven - the coverage is 100%.

Each alarm the analyzer reports has to be manually investigated to determine whether there is an error which has to be corrected, or whether it was just a false alarm. If all the alarms raised by an analysis have been proved to be false, then the proof of absence of runtime errors is completed. This could be checked manually, but the problem is that such a human analysis is error-prone and time consuming, especially since there might be interdependencies between the false alarms and in some cases deviations from the C standard may be willingly accepted. Therefore the number of alarms should be reduced to zero, since then the absence of runtime errors is automatically proved by the analyzer run.

To that end, it is important that the analyzer is precise, i.e., produces only few false alarms. This can only be achieved by a tool that can be "specialized" to a class of properties for a family of programs. Additionally the analyzer must be parametric enough for the user to be able to fine-tune the analysis of any particular program of the family. General software tools not amenable to specialization usually report a large number of false alarms which is the reason why such tools are only used in industry to detect runtime errors, and not to prove their absence.

Additionally the analyzer must provide flexible annotation mechanisms to communicate external knowledge to the analyzer. Only by a combination of high analyzer precision and support for semantic annotations the goal of *zero false alarms* can be achieved. A prerequisite is that users get enough information to understand the cause of an alarm so that they can either fix the bugs or supply the missing semantic information, e.g., about sensor input ranges.

3.4 Tool Support

The absence of non-functional errors from the categories listed above can be formally proven by Abstract Interpretation based static analysis tools. Examples are aiT [SLH+05] for worst-case execution time analysis, StackAnalyzer [FHF07] for stack usage analysis and Astrée [DS07] for runtime error analysis. For a more comprehensive overview of static analysis tools see [WEE+08] and [CCF+07].

Since runtime error analysis targets unspecified behaviors which are not covered by the language semantics the static analysis necessarily has to be based on the source code.

In contrast, for safe WCET and stack usage analysis it is important to work on fully linked *binary code*, i.e., here the static analysis is not based on the source code but on the executable code. In general, neither a sound stack usage analysis nor a sound WCET analysis

can be based on the source code. The compiler has a significant leeway to generate machine code from the source code given which can result in significant variations of stack usage and execution time. Moreover, for precise results it is important to perform a whole-program analysis after the linking stage. Lastly, for timing analysis precisely analyzing cache and pipeline behavior is imperative to obtain precise time bounds. This is only possible when analyzing machine instructions with known addresses of memory accesses.

Over the last few years, a more or less standard architecture for timing analysis tools has emerged [Erm03, FHL+01]. It neither requires code instrumentation nor debug information and is composed of three major building blocks:

- control-flow reconstruction and static analyses for control and data flow,
- micro-architectural analysis, computing upper bounds on execution times of basic blocks,
- path analysis, computing the longest execution paths through the whole program.

The data flow analysis of the first block also detects infeasible paths, i.e., program points that cannot occur in any real execution. This reduces the complexity of the following micro-architectural analysis. There, basic block timings are determined using an abstract processor model (*timing model*) to analyze how instructions pass through the pipeline taking cache-hit or cache-miss information into account. This model defines a cycle-level abstract semantics for each instruction's execution yielding in a certain set of final system states. After the analysis of one instruction has been finished, these states are used as start states in the analysis of the successor instruction(s). Here, the timing model introduces non-determinism that leads to multiple possible execution paths in the analyzed program. The pipeline analysis has to examine all of these paths. In general, the availability of safe worst-case execution time bounds depends on the predictability of the execution platform. Especially multi-core architectures may exhibit poor predictability because of essentially non-deterministic interferences on shared resources which can cause high variations in execution time. [CFG+10] gives a more detailed overview and suggests example configurations for available multi-cores to support static timing analysis.

This analysis architecture can also be applied to stack usage analysis. While for timing analysis the entire machine state has to be modelled, for stack usage analysis only stack pointer manipulations have to be covered.

4 Non-Functional Software Requirements in the ISO-26262

In this section we focus on [ISO11b], which specifies the requirements for product development at the software level for automotive applications. It covers the typical software development phases: requirements for initiation of software development, specification of software safety requirements, software architectural design, software unit design and implementation, software unit testing, software integration and testing, and verification of

software safety requirements. Non-functional software safety requirements are addressed in all these stages along the V-Model.

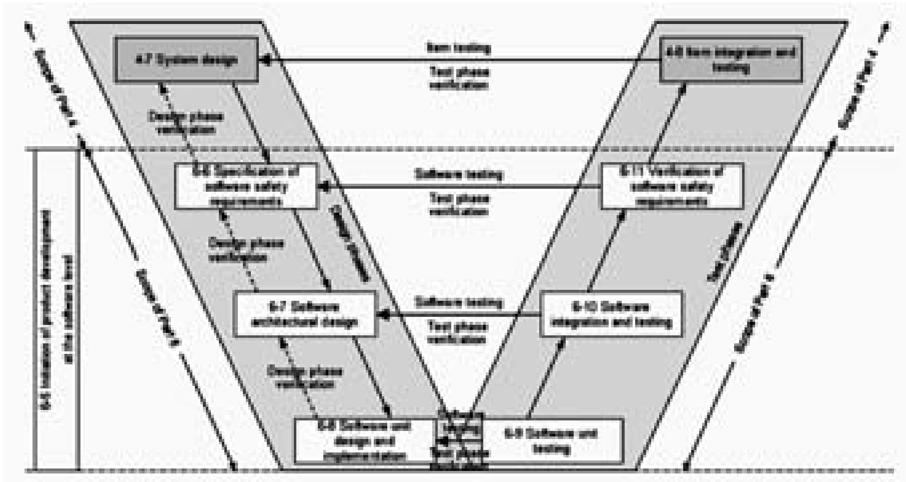

Figure 1: V-Model in the ISO-26262

4.1 Basic Requirements

The ability to handle non-functional program properties is one determining factor for selecting a suitable modelling or programming language (cf. [ISO11b], Sec. 5.4.6). The standard requires that real-time software and runtime error handling must be supported. It also states that "criteria that are not sufficiently addressed by the language itself shall be covered by the corresponding guidelines or by the development environment". Table 1 of [ISO11b] suggests the use of language subsets to exclude language constructs which could result in unhandled runtime errors. However, for typical embedded programming languages like C or C++, runtime errors can be caused by any pointer or array access, by arithmetic computations, etc. So in this case the absence of runtime errors has to be ensured by appropriate tools as a part of the development environment. Also timing and stack behavior is not captured in current programming language semantics and has to be addressed by specific tools.

4.2 Safety Requirements

In general the specification of the software safety requirements considers constraints of the hardware and the impact of these constraints on the software. Among others, safety requirements apply to functions that enable the system to achieve or maintain a safe state, and

to functions related to performance or time-critical operations. The standard explicitly lists some requirements which are part of the software safety, including the hardware-software interface specification, the relevant requirements of the hardware design specification, and the timing constraints. Hardware- or configuration-related errors (e.g., stack overflows and runtime errors like erroneous pointer manipulations) can cause globally unpredictable behavior affecting all safety functions and thus have to be taken into account. Timing constraints include the response time at the system level with derived timing properties like the worst-case execution time.

4.3 Software Architectural Design

The architectural design has to be able to realize the software safety requirements. The requirements listed in [ISO11b] include verifiability, feasibility for the design and implementation of the software units, and the testability of the software architecture during integration testing. In other words, the *predictability* of the system is one of the basic design criteria since predictability of the software as executed on the selected hardware is a precondition both for verifiability and for testability. Sec. 7.4.17 of [ISO11b] explicitly demands that "an upper estimation of required resources for the embedded software shall be made, including the execution time, and the storage space". Thus, upper bounds of the worst-case execution time and upper bounds of the stack usage are a fixed part of the architectural safety requirements. The importance of timing is is also reflected by the fact that "appropriate scheduling properties" are highly recommended for all ASIL levels as a principle for software architecture design. All existing schedulability algorithms assume upper bounds on the worst-case execution time to be known, as well as interferences on task switches either to be precluded or predictable. Thus the availability of safe worst-case execution and response times belong to the most basic scheduling properties.

Software architectural design requirements also explicitly address the interaction of software components. "Each software component shall be developed in compliance with the hightest ASIL of any requirements allocated to it." Furthermore, "all of the embedded software shall be treated in accordance with the highest ASIL, unless the software components meet the criteria for coexistence [...]" (cf. [ISO11b], Sec. 7.4.9.-7.4.10). Freedom of interference is an essential criterion for coexistence. Here all considerations of Sec. 3.2 apply. Freedom of interference also is addressed by Annex D of [ISO11b]. It discusses timing properties like worst-case execution time or scheduling characteristics as well as memory constraints. For memory safety, corruption of content, as well as read or write accesses to memory allocated to another software elements have to be excluded. Such accesses can be caused by stack overflows or runtime errors like erroneous pointer manipulations and dereferences. As a technique to show the absence of memory faults, Annex D lists static analysis of memory accessing software.

Table 6 of [ISO11b] lists the methods for verification of the software architectural design. Control flow analysis and data flow analysis are recommended for ASIL-A and ASIL-B and highly recommended for ASIL-C and ASIL-D; formal verification is recommended for ASIL-C and ASIL-D. This can be done separately for the modelling and the imple-

mentation level. However, with model-based code generation there is a synergy between model-level and implementation-level analysis (cf. Sec. 4.4). Since the semantics of the model is given by the generated implementation, source or binary code analyses can be used to verify the architectural design by propagating analysis results from the implementation level to the modelling level. This is well supported by static analysis techniques (cf. Sec. 5).

4.4 Software Unit Design, Implementation, and Testing

Chap. 8 of [ISO11b] defines the three goals of software unit design and implementation as: specification of the software units in accordance with design and safety requirements, implementation of the software units, and static verification of design and implementation of the software units. Thus, static verification plays a very prominent role in the design and implementation stage; it should always precede dynamic testing which should focus on properties not statically verified.

Table 9 lists the methods for verification of software unit design and implementation, including formal verification, control flow analysis, data flow analysis, static code analysis and semantic code analysis. As detailed in Sec. 2 all these techniques can be considered as aspects of general static analysis. Data and control flow analysis is a natural part of semantics-based static analyzers. With sound static analyzers proofs of data and control flow behavior can be obtained; they can be counted to the formal methods. A sound static analyzer for runtime error analysis like Astrée also provides safe data and control flow analysis without additional effort. The mentioned static analysis techniques are (highly) recommended for all ASIL levels.

In the structure of the ISO-26262 there is a differentiation between implementation, addressed in Chap. 8, and testing, addressed in Chap. 9. The absence of runtime errors (division by zero, control and data flow errors, etc) is considered as a robustness property (Sec. 8.4.4) which has to be ensured during implementation. Resource usage contraints like timing or stack consumption are addressed in Chap. 9 of [ISO11b] (Software Unit Testing). This distinction corresponds to the boundary between source code and binary code; apparently the underlying assumption is that source code can be analyzed and binary code has to be tested, which does not account for static analyses working at the binary code level. The requirements become consistent again when static analysis techniques are counted to the testing methods. Actually static analysis can be seen as an exhaustive testing method providing full coverage – a view shared by related safety standards like DO-178B [Rad], IEC-61508 [IEC10], and EN-50128 [CEN09].

4.5 Software Integration and Testing

The software integration phase has to consider functional dependences and the dependences between software integration and hardware-software integration. Again the non-

functional software properties have to be addressed; robustness has to be demonstrated which includes the absence of runtime errors, and it has to be demonstrated that there are sufficient resources to support the functionality which includes timing and stack usage (Sec. 10.4.3). Like in the unit testing stage, the ISO-26262 points out typical limitations of dynamic test and measuring techniques: when dynamic tests are made the test coverage has to be taken into account and it has to be shown that code modification and instrumentation does not affect the test results (cf. Sec. 9.4.5.-9.4.6; 10.4.5-10.4.6). It is important to perform WCET analysis in the unit testing stage to get early feedback of the timing behavior of the software components. However, it has to be noted that since the WCET depends on the memory addresses in the program and can be influenced by linking decisions, it also has to be applied in the integration stage on the final executable. The same considerations apply to interactions or interferences between software components.

5 Static Analysis Tools in the Development Process

Static analysis tools compute information about the software under analysis without actually executing it. No testing on physical hardware is required. As a consequence, static analyzers can be conveniently integrated in automatic build or testing systems in a fashion similar to the compilers.

Furthermore, it is straightforward to couple static code analysis tools with other development tools, e.g., with model-based code generators. After generating the code from the model, the static analyses can be automatically invoked from the modelling level. Also their results can be made available at the modelling level. Examples for such tool couplings exist between the static analyzers aiT, StackAnalyzer, Astrée and model-based code generators like dspace TargetLink, Esterel SCADE [ET], or ETAS ASCET. Industry partners of the European FP7 project INTERESTED where a complete integrated timing tool chain has been developed report reductions in overall project effort between 20% and over 50% [Int11].

Also tool couplings between different timing analysis tools can be beneficial. One example is a tool coupling between the aiT WCET Analyzer and the scheduling analysis tool SymTA/S [HHJ+05]. Here the goal is to provide a seamless framework for timing analysis at the code and system level providing both worst-case execution time and worst-case response time bounds. The advantage of such tool couplings is that they make it easy to use the static analysis during software development and not only in the final validation and verification stage. Defects can be detected early, so that late-stage integration problems can be avoided.

Many safety standards, including the ISO-26262, require development tools to be qualified, according to the criticality level of the tool usage. The qualification of static analysis tools can be automated to a large degree by dedicated *Qualification Support Kits* (QSKs). They consist of a report package and a test package. The report package lists all functional requirements and contains a verification test plan describing one or more test cases to check each functional requirement. The test package contains an extensible set of test

cases and a scripting system to automatically execute all test cases and evaluate the results. The generated reports can be submitted to the certification authority as part of the certification package. Additional Qualification Support Life Cycle Data (QSLCD) reports may be necessary to demonstrate the compliance of the tool development process to the standards for safety critical software development. QSKs and QSLCDs enable successful tool qualification for all safety standards up to the highest tool confidence levels.

6 Summary

The norm ISO-26262 requires to identify functional and non-functional safety hazards and to demonstrate that the software does not violate the relevant safety goals. Critical non-functional safety hazards can be caused by violations of timing constraints in real-time software and software malfunctioning due to runtime errors or stack overflows.

For such non-functional properties dynamic test and measurement techniques are often inappropriate since it is very hard to determine a safe test end criterion and typically no full test coverage can be obtained. One technique which achieves full control and data coverage and produces results valid for all program runs with all inputs is Abstract Interpretation, a semantics-based methodology for static program analysis. The term static analysis is used to describe a variety of program analysis techniques. We have given an overview of the different categories of static analyzers and outlined the principles of static analyzers for non-functional program properties. Abstract Interpretation-based tools are in industrial use that can prove the absence of stack overflows, runtime errors, and which can compute safe upper bounds on the worst-case execution time. The qualification of these tools as required by safety standards like the ISO-26262 can be automated to a large extend by dedicated Qualification Support Kits.

The ISO-26262 ranks the static verification of program properties among the prominent goals of the software development process. We have detailed the requirements of the ISO-26262 for non-functional program properties and discussed the role of static analyzers in the ISO-26262.

Acknowledgements

The work presented in this paper has been supported by the European FP6 project INTEREST, the European FP7 projects INTERESTED and PREDATOR, the ITEA2 project ES_PASS, and the ARTEMIS project MBAT.

References

[CC77] Patrick Cousot and Radhia Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proceedings of the 4th ACM Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, California, 1977.

[CCF+07] Patrick Cousot, Radhia Cousot, Jérôme Feret, Antoine Miné, Laurent Mauborgne, David Monniaux, and Xavier Rival. Varieties of Static Analyzers: A Comparison with ASTRÉE. In *First Joint IEEE/IFIP Symposium on Theoretical Aspects of Software Engineering, TASE 2007*, pages 3–20. IEEE Computer Society, 2007.

[CEN09] CENELEC DRAFT prEN 50128. Railway applications – Communication, signalling and processing systems – Software for railway control and protection systems, 2009.

[CFG+10] Christoph Cullmann, Christian Ferdinand, Gernot Gebhard, Daniel Grund, Claire Maiza (Burguière), Jan Reineke, Benoît Triquet, Simon Wegener, and Reinhard Wilhelm. Predictability Considerations in the Design of Multi-Core Embedded Systems. *Ingénieurs de l'Automobile*, 807:26–42, 2010.

[DS07] D. Delmas and J. Souyris. ASTRÉE: from Research to Industry. In *Proc. 14th International Static Analysis Symposium (SAS2007)*, number 4634 in LNCS, pages 437–451, 2007.

[ECH+01] D. Engler, D. Y. Chen, S. Hallem, A. Chou, and B. Chelf. Bugs as Deviant Behavior: A General Approach to Inferring Errors in Systems Code. In *18th ACM Symp. on Operating Systems Principles*, 2001.

[Erm03] Andreas Ermedahl. *A Modular Tool Architecture for Worst-Case Execution Time Analysis*. PhD thesis, Uppsala University, 2003.

[ET] Esterel Technologies. SCADE Suite.
`http://www.esterel-technologies.com/products/scade-suite`.

[Fer97] Christian Ferdinand. *Cache Behavior Prediction for Real-Time Systems*. PhD thesis, Saarland University, 1997.

[FHF07] Christian Ferdinand, Reinhold Heckmann, and Bärbel Franzen. Static Memory and Timing Analysis of Embedded Systems Code. In Perry Groot, editor, *Proceedings of the 3rd European Symposium on Verification and Validation of Software Systems (VVSS 2007) on March 23, 2007 at Eindhoven, The Netherlands*, volume 07-04 of *TUE Computer Science Reports*, pages 153–163, Eindhoven, 2007.

[FHL+01] Christian Ferdinand, Reinhold Heckmann, Marc Langenbach, Florian Martin, Michael Schmidt, Henrik Theiling, Stephan Thesing, and Reinhard Wilhelm. Reliable and Precise WCET Determination for a Real-Life Processor. In *Proceedings of EMSOFT 2001, First Workshop on Embedded Software*, volume 2211 of *Lecture Notes in Computer Science*, pages 469–485. Springer-Verlag, 2001.

[Hat95] L. Hatton. Safer C: Developing for High-Integrity and Safety-Critical Systems. *McGraw-Hill*, 1995.

[HHJ+05] R. Henia, A. Hamann, M. Jersak, R. Racu, K. Richter, and R. Ernst. System Level Performance Analysis – the SymTA/S Approach. *IEEE Proceedings on Computers and Digital Techniques*, 152(2), March 2005.

[Hol02] G. Holzmann. UNO: Static Source Code Checking for User- Defined Properties. In *6th World Conf. on Integrated Design and Process Technology (IDPT02)*, 2002.

[IEC10] IEC 61508. Functional safety of electrical/electronic/programmable electronic safety-related systems, 2010.

[Int11] INTERESTED Project Final Report (Public Version). `http://www.interested-ip.eu/`, 2011.

[ISO11a] ISO/FDIS 26262. Road vehicles – Functional safety, 2011.

[ISO11b] ISO/FDIS 26262. Road vehicles – Functional safety – Part 6: Product development at the software level, 2011.

[KFH+11] D. Kästner, C. Ferdinand, R. Heckmann, M. Jersak, and P. Gliwa. An Integrated Timing Analysis Methodology for Real-Time Systems. *Embedded World Congress*, 2011.

[Klo] Klocwork. Klocwork K7^{TM}. `http://www.klocwork.com`.

[LE01] D. Larochelle and D. Evans. Statically Detecting Likely Buffer Overflow Vulnerabilities. In *2001 USENIX Security Symposium, Washington, D.C.*, 2001.

[Lea96] J.L. Lions et al. ARIANE 5, Flight 501 Failure. *Report by the Inquiry*, 1996.

[Min04] A. Miné. *Weakly Relational Numerical Abstract Domains*. PhD thesis, École Polytechnique, Palaiseau, France, December 2004. `http://www.di.ens.fr/~mine/these/these-color.pdf`.

[Mot04] The Motor Industry Software Reliability Association. *Guidelines for the Use of the C Language in Critical Systems*, October 2004. ISBN 0-9524156-2-3.

[NAS11] NASA Engineering and Safety Center. Technical Support to the National Highway Traffic Safety Administration (NHTSA) on the Reported Toyota Motor Corporation (TMC) Unintended Acceleration (UA) Investigation, 2011.

[Rad] Radio Technical Commission for Aeronautics. RTCA DO-178B. Software Considerations in Airborne Systems and Equipment Certification.

[SLH+05] Jean Souyris, Erwan Le Pavec, Guillaume Himbert, Victor Jégu, Guillaume Borios, and Reinhold Heckmann. Computing the Worst Case Execution Time of an Avionics Program by Abstract Interpretation. In *Proceedings of the 5th Intl Workshop on Worst-Case Execution Time (WCET) Analysis*, pages 21–24, 2005.

[The04] Stephan Thesing. *Safe and Precise WCET Determinations by Abstract Interpretation of Pipeline Models*. PhD thesis, Saarland University, 2004.

[WEE+08] Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Frank Mueller, Isabelle Puaut, Peter Puschner, Jan Staschulat, , and Per Stenstroem. The Determination of Worst-Case Execution Times —Overview of the Methods and Survey of Tools. *ACM Transactions on Embedded Computing Systems (TECS)*, 2008.